CMPS 6100 Lab 10

In this lab, you will turn your basic chat program into a a chat application that can handle multiple clients, allowing all to chat in a shared chat space. In the process, you will refactor your client and server to be multi-threaded to allow for smooth handling of all users.

Refer back to the README.md for instruction on git and how to submit properly to get all the points you've earned.

Let's build chat for the class

Your task is to turn this basic client/server chat program into one where the server can accept and handle multiple clients simultaneously.

The server will be a central server. That is, Every client will connect to it. If the server receives a message from any client, it will send it to all other clients (excluding the client that originally sent it.)

The Client

The client is simple and dumb.

At start, your client will ask the user for a username and then connect to the server.

The client will send the username as its first message to the server (no trailing new line characters).

It will then receive and print a welcome message from the server.

Then it will start doing two things:

1. Receiving and printing messages from the server
2. Reading in and sending messages from the user to the server

Since the order that messages will be received and sent is nondeterministic, these two tasks will have to be in separate threads.

In the read-from-server thread (the one that receives and prints messages from the server), the client will enter a loop, printing every message that is received from the server. Here, the client is dumb. It simply prints every message received.

In the client thread (the one that reads messages from the user and sends them to the server), again, the client is dumb. It reads in a message from the user, and simply sends it to the server. The only processing it does is to remove the ending new line character before sending. For simplicity, your client does NOT need to emit a prompt.

The Server

The server is more involved.

Requirements:

- Your server must allow multiple clients to connect simultaneously.
  - You server must set a limit (your choice) on the number of connected clients. Define this as a constant at the top of the server.
- When a client connects, the server must send that client a welcome message:
  - If there are no other clients connected, the welcome message should be :

    ```
    Welcome to chat!
    There are no other users currently logged in.
    ```

  - If there are other clients connected, for example, `"alice"`, `"bob"`, and `"eve"`, the message should be:

    ```
    Welcome to chat!
    Connected users:
    - alice
    - bob
    - eve
    ```

    The order that the users are listed in may be arbitary.

  - To all other connected clients, the server must send a message:

    `"<username> has connected to chat."`

    Where username is the name of the user who connected. For example, when alice connects, all other clients receive the message:

    `"alice has connected to chat."`

- If a message, `"logout"` is received from a client, then the server should close that client's socket, remove it from the list of connected clients, and send a message to all remaining clients stating that client has disconnected.

  e.g,

  `"alice has disconnected from chat."`

- Otherwise, when a message is received from any client, that message is concatenated to a string of the form `"username: "`, and this new concatenated string is sent to all other clients.

  - For example, if `"alice"` sends the message `"hello"`, then the message:

    `"alice: hello"`

    is sent to all other clients

- Your server may print anything it likes to `stdout`. As a sanity check, we recommend at a minimum printing a log of client connections and disconnections. Seeing the messages that clients send to each other can also be useful.

Tips

- Since clients may connect and send messages nondeterministically, the server, like the client, must be multithreaded. Every client should be handled in its own thread.
- Tracking connected clients will be crucial since every message must be sent to all other clients. We recommend defining a struct for client information (namely its socket and username), and creating an array of client structs.
  - How to add and remove clients from this array when a new client connects or disconnects is an important consideration. We recommend modelling this after the `jobs` array from `tsh`.
  - When modifying this array, you will want to be very keen to avoid race conditions. If you initialize a global lock for the server and always grab it before and release it after updating or accessing the client list, you should avoid all race conditions.
- When developing, you will likely start and stop your client/server app many times. Best practice is to stop all clients connected to the server before stopping the server. If you don't, you may get a `server: bind error: Address already in use`
- If you get a `server: bind error: Address already in use`, your server likely didn't exit correctly (see above, but could also happen for other reasons). You can try the following command

  ```
  $ lsof -n -i :PORTNUM
  ```

(where `PORTNUM` is the port number you have chosen) to get the PID of the process bound to that port. Once you know the PID, if you own that process, you can kill it with the command

```
$ kill PID
```

Alternatively, choose a new port number for your chat program.

Compiling your client and server

Since your client and server a multithreaded, you will need to modify your compilation commands in order to compile them:

```
$ gcc -pthread server.c -o server
$ gcc -pthread client.c -o client
```

Reference Executables

Reference executables `client_ref` and `server_ref` have been provided for you. Run these programs to resolve any questions you have about how your application should behave.

You can and should have multiple copies of `client_ref` connect to the same instance of `server_ref`.

Resources

- OSTEP: Ch. 27 Interlude: Thread API
- CNASA: Ch. 1.4: Software
- Harvard's CS50 Manual Page
- Harvard's CS50 Duck AI
- Stanford's Essential C